

Trabalho Prático 2

Restaurante

**Compreensão dos mecanismos associados à execução
e sincronização de processos e *threads***

Licenciatura em Engenharia Informática
UC 40381 - Sistemas Operativos
Ano letivo 2018/2019

Trabalho realizado por:
88808 - João Miguel Nunes de Medeiros e Vasconcelos
88886 - Tiago Carvalho Mendes

Aveiro, 11 de janeiro de 2019

Índice

Introdução.....	3
Apresentação do problema.....	4
Análise do código fornecido.....	5
Implementação da solução.....	9
Execução do programa.....	27
Conclusão.....	30
Referências.....	30

Introdução

O presente documento destina-se a descrever detalhadamente a abordagem utilizada para resolver o problema proposto como segundo trabalho prático da unidade curricular de **Sistemas Operativos da Universidade de Aveiro**.

O principal objetivo deste segundo trabalho prático passa não só pela compreensão dos mecanismos associados à execução e sincronização de processos e *threads* bem como pela introdução da programação concorrente.

Para atingir tal objetivo, foi fornecido a todos os alunos na página da disciplina algum código fonte, servindo como ponto de partida para o desenvolvimento de uma aplicação utilizando a **linguagem de programação C** que simula um restaurante, aplicação esta que será analisada detalhadamente nas páginas seguintes.

O trabalho foi realizado pelos **dois alunos** identificados na capa deste documento, ambos com uma percentagem de **50% de contribuição**.

De modo a suportar a realização do trabalho, foi criado **um repositório GIT privado na plataforma code.ua.pt**.

Apresentação do problema

A primeira tarefa realizada pelo nosso grupo foi a de analisar cuidadosamente a apresentação do problema no guião disponibilizado na página da disciplina. Como foi dito anteriormente, era pedido nesse guião o desenvolvimento de uma aplicação na linguagem de programação C de modo a simular um restaurante. Este restaurante tem portanto **4 entidades intervenientes**, nomeadamente:

- No máximo 16 **Grupos** (*Group*)
- Um **Recepção** (*Receptionist*)
- Um **Empregado** de mesa (*Waiter*)
- Um **Cozinheiro** (*Chef*)

Estas 4 entidades serão portanto **processos independentes** criados no início da execução do programa principal, sendo a sua sincronização realizada através de **semáforos e acessos à memória partilhada**. Estes processos deverão estar ativos **apenas quando for necessário**, devendo **bloquear** sempre que têm de esperar por algum evento.

No entanto, devem ser seguidas algumas regras de modo a sincronizar da melhor forma todas as entidades envolvidas, simulando com eficácia o modo de funcionamento de um restaurante no contexto real. **As regras são as seguintes:**

- O **Restaurante** tem apenas 2 mesas.
- Cada **Grupo** deve dirigir-se em primeiro lugar ao **Recepção**.
- O **Recepção**, conforme a disponibilidade das 2 mesas, irá indicar ao **Grupo** a mesa a ocupar ou se deve esperar.
- Após obter mesa, o **Grupo** pede comida ao **Empregado**, ficando à espera da comida.
- O **Empregado** leva o pedido ao **Cozinheiro**, ficando a aguardar a preparação da comida.
- O **Cozinheiro** recebe o pedido, prepara a refeição, e avisa o **Empregado** assim que estiver pronta.
- O **Empregado** leva a comida à mesa do **Grupo** correspondente.
- O **Grupo** inicia a refeição e, após terminar, contacta o **Recepção** de modo a pagar a conta e sair.
- O **Recepção** recebe o pagamento e, se existirem **Grupos** à espera, atribui ao que está há mais tempo a mesa que acabou de ficar disponível.

Tendo em conta estas regras, nas páginas seguintes será analisado todo o código fornecido e desenvolvido de modo a um correto funcionamento da aplicação.

Análise do código fonte fornecido

Após uma análise do guião disponibilizado, o nosso grupo analisou o código fonte fornecido na página da disciplina. O ficheiro disponibilizado contém os seguintes diretórios:

- `semaphore_restaurant/doc` - diretório com documentação sobre a aplicação
- `semaphore_restaurant/run` - diretório com todos os executáveis da aplicação
- `semaphore_restaurant/src` - diretório com todo o código fonte disponibilizado

Começando pelos estados que cada entidade apresentada na página anterior pode assumir, tendo em conta a ação que está a desempenhar, tem-se o seguinte:

Ficheiro src/probConst.h:

```
#define MAXGROUPS 16
    maximum number of groups
#define NUMTABLES 2
    number of tables
#define MAXCOOK 100
    controls time taken to cook
#define STARTDEV 4
    controls start time standard deviation
#define EATDEV 4
    controls eat time standard deviation
```

Figura 1 - Algumas Macros importantes

```
#define TABLEREQ 1
    id of table request (group->receptionist)
#define BILLREQ 2
    id of bill request (group->receptionist)
#define FOODREQ 3
    id of food request (group->waiter)
#define FOODREADY 4
    id of food ready (chef->waiter)
```

Figura 2 - Possíveis tipos de pedido

Sistemas Operativos - Trabalho Prático 2

#define GOTOREST 1	group initial state
#define ATRECEPTION 2	client is waiting at reception or waiting for table
#define FOOD_REQUEST 3	client is requesting food to waiter
#define WAIT_FOR_FOOD 4	client is waiting for food
#define EAT 5	client is eating
#define CHECKOUT 6	client is checking out
#define LEAVING 7	client is leaving

Figura 3 - Estados possíveis dos *Grupos*

#define WAIT_FOR_ORDER 0	chef waits for food order
#define COOK 1	chef is cooking
#define REST 2	chef is resting

Figura 4 - Estados possíveis do *Cozinheiro*

#define WAIT_FOR_REQUEST 0	waiter or receptionist waits for request
----------------------------	--

Figura 5 - Estado possível do *Receptionista* ou do *Empregado*

#define INFORM_CHEF 1	waiter takes food request to chef
#define TAKE_TO_TABLE 2	waiter takes food to table

Figura 6 - Estados possíveis do *Empregado*

#define ASSIGNTABLE 1	receptionist performs checkin
#define RECVPAY 2	receptionist receives payment

Figura 7 - Estados possíveis do *Receptionista*

Ficheiro src/probDataStruct.h:

Este ficheiro contém a definição de todas as estruturas de dados internas que especificam os estados das entidades intervenientes. Não entraremos em muitos detalhes sobre estas estruturas nesta página, visto que serão exploradas detalhadamente mais à frente.

probDataStruct.h File Reference

Problem name: Restaurant. [More...](#)

```
#include <stdbool.h>
#include "probConst.h"
```

[Go to the source code of this file.](#)

Data Structures

struct request	Definition of requests to receptionist and waiter. More...
struct STAT	Definition of state of the intervening entities data type. More...
struct FULL_STAT	Definition of full state of the problem data type. More...

Figura 8 - Referências do ficheiro *probDataStruct.h*

Sistemas Operativos - Trabalho Prático 2

Ficheiro sharedDataSync.h

Neste ficheiro, encontram-se definidos os formatos da memória partilhada, que representa o estado completo do problema (**FULL_STAT fSt**), e a identificação dos diferentes semáforos, que são responsáveis pela sincronização entre as entidades intervenientes na aplicação.

```
26 typedef struct
27     { /* \brief full state of the problem */
28         FULL_STAT fSt;
29
30         /* semaphores ids */
31         /* \brief identification of critical region protection semaphore - val = 1 */
32         unsigned int mutex;
33         /* \brief identification of semaphore used by receptionist to wait for groups - val = 0 */
34         unsigned int receptionistReq;
35         /* \brief identification of semaphore used by groups to wait before issuing receptionist request - val = 1 */
36         unsigned int receptionistRequestPossible;
37         /* \brief identification of semaphore used by waiter to wait for requests - val = 0 */
38         unsigned int waiterRequest;
39         /* \brief identification of semaphore used by groups and chef to wait before issuing waiter request - val = 1 */
40         unsigned int waiterRequestPossible;
41         /* \brief identification of semaphore used by chef to wait for order - val = 0 */
42         unsigned int waitOrder;
43         /* \brief identification of semaphore used by waiter to wait for chef - val = 0 */
44         unsigned int orderReceived;
45         /* \brief identification of semaphore used by groups to wait for table - val = 0 */
46         unsigned int waitForTable[MAXGROUPS];
47         /* \brief identification of semaphore used by groups to wait for waiter acknowledge - val = 0 */
48         unsigned int requestReceived[NUMTABLES];
49         /* \brief identification of semaphore used by groups to wait for food - val = 0 */
50         unsigned int foodArrived[NUMTABLES];
51         /* \brief identification of semaphore used by groups to wait for payment completed - val = 0 */
52         unsigned int tableDone[NUMTABLES];
53
54     } SHARED_DATA;
```

Figura 9 - Referências da estrutura **SHARED_DATA**

Macros

#define SEM_NU (7 + sh->fSt.nGroups + 3*NUMTABLES) number of semaphores in the set
#define MUTEX 1
#define RECEPTIONISTREQ 2
#define RECEPTIONISTREQUESTPOSSIBLE 3
#define WAITERREQUEST 4
#define WAITERREQUESTPOSSIBLE 5
#define WAITORDER 6
#define ORDERRECEIVED 7
#define WAITFORTABLE 8
#define FOODARRIVED (WAITFORTABLE+sh->fSt.nGroups)
#define REQUESTRECEIVED (FOODARRIVED+NUMTABLES)
#define TABLEDONE (REQUESTRECEIVED+NUMTABLES)

Figura 10 - Macros dos semáforos apresentados na Figura 9

Implementação da solução

Após uma cuidada análise de todo o código fonte disponibilizado, o nosso grupo estruturou uma possível implementação da solução final. Como foi visto nas páginas anteriores, esta possível implementação tem por base a utilização de **semáforos e memória partilhada**. Existem ao todo $7 + (\#Grupos) + 3 * (\#Mesas)$ semáforos, como mostrado na *Figura 10*, sendo que um deles funciona em **exclusão mútua (mutex)**, de modo a ser possível aceder à memória partilhada.

Como possuímos 4 entidades com **necessidade de partilhar informações** entre si, a utilização de semáforos torna-se **essencial**, visto que é necessário garantir que não existem **conflitos entre as mudanças de estado** das mesmas. Concretamente, podemos controlar **quem acede à memória partilhada**, garantindo que apenas uma entidade se encontra **de cada vez** na mesma. **Exemplificando**, suponha-se que um determinado **Grupo** pretende efetuar um pedido de atribuição da conta a pagar ao **Recepcionista**. Utilizando **semáforos**, este pedido só será efetuado assim que o **Recepcionista** estiver disponível, e esse **Grupo** só poderá sair quando o **Recepcionista** confirmar o pagamento. **Todas estas situações serão explicadas nas páginas seguintes**.

Assim sendo, existem 4 ficheiros no diretório `semaphore_restaurant/src incompletos`, cada um definindo as operações que cada uma das 4 entidades realiza. Esses ficheiros são os seguintes:

- `semSharedMemChef.c`
- `semSharedMemWaiter.c`
- `semSharedMemReceptionist.c`
- `semSharedMemGroup.c`

As zonas de código a desenvolver já se encontram previamente assinaladas, bem como as **regiões críticas** dentro de cada função. Sempre que se quiser aceder a uma zona de memória partilhada, é necessário fazer primeiro a operação **semDown()** do semáforo **mutex** e no final a operação **semUp()**, imprimindo o programa uma mensagem de erro se algo correr mal. De modo a uma melhor compreensão do código a desenvolver, todas as funções incompletas contêm comentários explicativos sobre o fluxo normal de execução.

Sistemas Operativos - Trabalho Prático 2

Tendo em conta tudo o que foi dito na página anterior, a seguinte tabela foi criada com o intuito de melhorar a visualização do problema, identificando as zonas de código onde era necessário atualizar cada semáforo.

Semáforo	Entidade		Ação		Função	
	Up()	Down()	Up()	Down()	Up()	Down()
mutex	Todas	Todas	Sair da região crítica	Entrar na região crítica	Quase todas	Quase todas
receptionistReq	Group	Receptionist	Desbloqueia o sem assim que efetua um pedido ao Recepçionista	Bloqueia o sem à espera de um novo pedido do Grupo	checkInAtReception() checkOutAtReception()	waitForGroup()
receptionRequest_Possible	Receptionist	Group	Desbloqueia o sem assim que estiver disponível para novos pedidos	Bloqueia o sem para averiguar a disponibilidade do Recepçionista	waitForGroup()	checkInAtReception() checkOutAtReception()
waiterRequest	Chef Group	Waiter	Desbloqueia o sem assim que efetua um pedido ao Empregado	Bloqueia o sem à espera de um novo pedido	processOrder() orderFood()	waitForClientOrChef()
waiterRequest_Possible	Waiter	Chef Group	Desbloqueia o sem assim que estiver disponível para novos pedidos	Bloqueia o sem para averiguar a disponibilidade do Empregado	waitForClientOrChef()	processOrder() orderFood()
waitOrder	Waiter	Chef	Desbloqueia o sem assinalando um pedido ao Chef	Bloqueia o sem à espera de um novo pedido do Empregado	informChef()	waitForOrder()
orderReceived	Chef	Waiter	Desbloqueia o sem assinalando que recebeu o pedido do Empregado	Bloqueia o sem à espera que o Chef receba o pedido	waitForOrder()	informChef()
waitForTable	Receptionist	Group	Desbloqueia o sem assim que estiver uma mesa disponível	Bloqueia o sem à espera que lhe seja atribuído uma mesa	provideTableOrWaitingRoom() receivePayment()	checkInAtReception()
requestReceived	Waiter	Group	Desbloqueia o sem assinalando que recebeu o pedido do Grupo	Bloqueia o sem à espera que o Empregado receba o pedido	informChef()	orderFood()
foodArrived	Waiter	Group	Desbloqueia o sem assinalando que a comida chegou à mesa	Bloqueia o sem à espera que o Empregado traga a comida	takeFoodToTable()	waitForFood()
tableDone	Receptionist	Group	Desbloqueia o sem assinalando que o pagamento foi completado	Bloqueia o sem assim que faz o pedido de pagamento	receivePayment()	checkOutAtReception()

Tabela 1 - Semáforos existentes e operações correspondentes

Com a *Tabela 1*, os semáforos existentes e as operações correspondentes em cada entidade foram **claramente identificados**, facilitando a implementação da solução. Nas páginas seguintes encontram-se explicações para todo o código desenvolvido, com capturas de ecrã ilustrativas.

Entidade Cozinheiro - Ficheiro *semSharedMemChef.c*

- Função *static void waitForOrder()*

Nesta função, o **Cozinheiro** espera pelo pedido de comida levado pelo **Empregado** através da operação *semDown* do semáforo *waitOrder*. Ao receber o pedido e **dentro da região crítica**, altera o seu estado para **COOK (1)**, atualiza o valor da flag *foodOrder = 0*, altera o valor da variável *lastGroup* para o **id do grupo** que fez o pedido e guarda o seu estado interno. Por fim, efetua a operação *semUp* do semáforo *orderReceived* assinalando que recebeu um pedido do **Empregado**.

```
126 static void waitForOrder ()
127 {
128
129     /* STUDENTS CODE */
130     if (semDown (semgid, sh->waitForOrder) == -1) {
131         perror ("error on the down operation for waitForOrder semaphore access (CH)");
132         exit (EXIT_FAILURE);
133     }
134     /* END OF STUDENTS CODE REGION */
135
136     /* enter critical region */
137     if (semDown (semgid, sh->mutex) == -1) {
138         perror ("error on the down operation for mutex semaphore access (CH)");
139         exit (EXIT_FAILURE);
140     }
141
142     /* STUDENTS CODE */
143     sh->fst.st.chefStat = COOK;
144     sh->fst.foodOrder = 0;
145     lastGroup = sh->fst.foodGroup;
146     saveState(nFic, &sh->fst);
147     /* END OF STUDENTS CODE REGION */
148
149     /* exit critical region */
150     if (semUp (semgid, sh->mutex) == -1) {
151         perror ("error on the up operation for mutex semaphore access (CH)");
152         exit (EXIT_FAILURE);
153     }
154
155     /* STUDENTS CODE */
156     if (semUp (semgid, sh->orderReceived) == -1) {
157         perror ("error on the up operation for orderReceived semaphore access (CH)");
158         exit (EXIT_FAILURE);
159     }
160     /* END OF STUDENTS CODE REGION */
161 }
```

Figura 11 - Função *waitForOrder()*

- Função **static void processOrder()**

Nesta função, o **Cozinheiro** demora algum tempo a preparar a comida, operação simulada através da função **usleep()**. Assim que estiver pronta, o **Cozinheiro** espera até que o **Empregado** se encontre disponível através da operação **semDown** do semáforo **waiterRequestPossible**. De seguida e dentro da **região crítica**, altera o tipo do pedido ao **Empregado** para **FOODREADY (4)**, informando a que grupo pertence a comida confecionada. O **Cozinheiro** altera o seu estado para **WAIT_FOR_ORDER (0)** e guarda o seu estado interno, ainda dentro da **região crítica**. Por fim, efetua a operação **semUp** do semáforo **waiterRequest**, informando o **Empregado** que a comida está pronta para ser levada para a mesa.

```
171 static void processOrder ()  
172 {  
173     usleep((unsigned int) floor ((MAXCOOK * random ()) / RAND_MAX + 100.0));  
174  
175     /* STUDENTS CODE */  
176     if (semDown (semgid, sh->waiterRequestPossible) == -1) {  
177         perror ("error on the down operation for waiterRequestPossible semaphore access (CH)");  
178         exit (EXIT_FAILURE);  
179     }  
180     /* END OF STUDENTS CODE REGION */  
181  
182     /* enter critical region */  
183     if (semDown (semgid, sh->mutex) == -1) {  
184         perror ("error on the down operation for mutex semaphore access (CH)");  
185         exit (EXIT_FAILURE);  
186     }  
187  
188     /* STUDENTS CODE */  
189     sh->fst.waiterRequest.reqType = FOODREADY;  
190     sh->fst.waiterRequest.reqGroup = lastGroup;  
191     sh->fst.st.chefstat = WAIT_FOR_ORDER;  
192     saveState(nFic, &sh->fst);  
193     /* END OF STUDENTS CODE REGION */  
194  
195     /* exit critical region */  
196     if (semUp (semgid, sh->mutex) == -1) {  
197         perror ("error on the up operation for mutex semaphore access (CH)");  
198         exit (EXIT_FAILURE);  
199     }  
200  
201     /* STUDENTS CODE */  
202     if (semUp (semgid, sh->waiterRequest) == -1) {  
203         perror ("error on the up operation for waiterRequest semaphore access (CH)");  
204         exit (EXIT_FAILURE);  
205     }  
206     /* END OF STUDENTS CODE REGION */  
207 }
```

Figura 12 - Função *processOrder()*

Entidade Empregado - Ficheiro *semSharedMemWaiter.c*

- Função *static request waitForClientOrChef()*

Nesta função, o **Empregado** altera o seu estado para **WAIT_FOR_REQUEST (0)** dentro da **região crítica**, guardando-o. De seguida, espera por um novo pedido por parte de um dos **Grupos** ou do **Cozinheiro** através da operação **semDown** do semáforo **waiterRequest**. Ao receber o pedido, guarda o tipo e o grupo do mesmo na variável **req**, que será retornada no final da função. Por fim, efetua a operação **semUp** do semáforo **waiterRequestPossible**, assinalando que se encontra disponível para receber novos pedidos.

```
137 static request waitForClientOrChef()
138 {
139     request req;
140
141     /* enter critical region */
142     if (semdown (semgid, sh->mutex) == -1) {
143         perror ("error on the down operation for mutex semaphore access (WT)");
144         exit (EXIT_FAILURE);
145     }
146
147     /* STUDENTS CODE */
148     sh->fst.st.waiterStat = WAIT_FOR_REQUEST;
149     saveState(nPic,&sh->fst);
150     /* END OF STUDENTS CODE REGION */
151
152     /* exit critical region */
153     if (semup (semgid, sh->mutex) == -1) {
154         perror ("error on the up operation for mutex semaphore access (WT)");
155         exit (EXIT_FAILURE);
156     }
157
158     /* STUDENTS CODE */
159     if (semdown (semgid, sh->waiterRequest) == -1) {
160         perror ("error on the down operation for waiterRequest semaphore access (WT)");
161         exit (EXIT_FAILURE);
162     }
163     /* END OF STUDENTS CODE REGION */
164
165     /* enter critical region */
166     if (semdown (semgid, sh->mutex) == -1) {
167         perror ("error on the down operation for mutex semaphore access (WT)");
168         exit (EXIT_FAILURE);
169     }
170
171     /* STUDENTS CODE */
172     req.reqGroup = sh->fst.waiterRequest.reqGroup;
173     req.reqType = sh->fst.waiterRequest.reqType;
174     /* END OF STUDENTS CODE REGION */
175
176     /* exit critical region */
177     if (semup (semgid, sh->mutex) == -1) {
178         perror ("error on the up operation for mutex semaphore access (WT)");
179         exit (EXIT_FAILURE);
180     }
181
182     /* STUDENTS CODE */
183     if (semup (semgid, sh->waiterRequestPossible) == -1) {
184         perror ("error on the up operation for waiterRequestPossible semaphore access (WT)");
185         exit (EXIT_FAILURE);
186     }
187     /* END OF STUDENTS CODE REGION */
188
189     return req;
190 }
191 }
```

Figura 13 - Função *waitForClientOrChef()*

- Função **static void informChef(int n)**

Nesta função, o **Empregado** sabe que tem um pedido de comida por parte de um dos **Grupos**. Portanto, dentro da **região crítica**, altera o seu estado para **INFORM_CHEF (1)**. Também altera não só o valor da *flag foodOrder = 1* mas também o valor da *flag foodGroup* para o *id* do **Grupo** que fez o pedido. Antes de sair da **região crítica**, guarda o seu estado. De seguida, leva o seu pedido ao **Cozinheiro** através da operação **setUp** do semáforo **waitOrder**. O **Empregado** informa ainda o **Grupo** correspondente que recebeu o pedido através da operação **setUp** do semáforo **requestReceived[assignedTable]** com o índice da mesa em que o **Grupo** se encontra. Por fim, efetua a operação **semDown** do semáforo **orderReceived** à espera que o **Cozinheiro** receba o pedido.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** está sentado, de modo a ser possível realizar a operação **setUp** do semáforo **requestReceived[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada** fora da **região crítica**.

```
202 static void informChef (int n)
203 {
204     int assignedTable;
205
206     /* enter critical region */
207     if (semDown (semgid, sh->mutex) == -1) {
208         perror ("error on the down operation for mutex semaphore access (WT)");
209         exit (EXIT_FAILURE);
210     }
211
212     /* STUDENTS CODE */
213     sh->fSt.st.waiterStat = INFORM_CHEF;
214     sh->fSt.foodOrder = 1;
215     sh->fSt.foodGroup = n;
216     assignedTable = sh->fSt.assignedTable[n];
217     saveState(nFic, &sh->fSt);
218     /* END OF STUDENTS CODE REGION */
219
220     /* exit critical region */
221     if (semUp (semgid, sh->mutex) == -1) {
222         perror ("error on the up operation for mutex semaphore access (WT)");
223         exit (EXIT_FAILURE);
224     }
225
226     /* STUDENTS CODE */
227     if (semUp (semgid, sh->waitOrder) == -1){
228         perror ("error on the up operation for waitOrder semaphore access (WT)");
229         exit (EXIT_FAILURE);
230     }
231
232     if (semUp (semgid, sh->requestReceived[assignedTable]) == -1) {
233         perror ("error on the up operation for requestReceived semaphore access (WT)");
234         exit (EXIT_FAILURE);
235     }
236
237     if (semDown (semgid, sh->orderReceived) == -1) {
238         perror ("error on the down operation for orderReceived semaphore access (WT)");
239         exit (EXIT_FAILURE);
240     }
241     /* END OF STUDENTS CODE REGION */
```

Figura 14 - Função *informChef()*

- Função **static void takeFoodToTable(int n)**

Nesta função, o **Empregado**, dentro da região crítica, altera não só o seu estado para **TAKE_TO_TABLE (2)** mas também o valor da flag **foodGroup** para o id do **Grupo** correspondente. De seguida, efetua a operação **setUp** do semáforo **foodArrived[assignedTable]** com o índice da mesa em que o **Grupo** correspondente se encontra, assinalando que a comida chegou à mesa e que o grupo pode começar a refeição.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** está sentado, de modo a ser possível realizar a operação **setUp** do semáforo **foodArrived[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada** fora da **região crítica**.

```
254 static void takeFoodToTable (int n)
255 {
256     int assignedTable;
257
258     /* enter critical region */
259     if (semDown (semgid, sh->mutex) == -1) {
260         perror ("error on the down operation for mutex semaphore access (WT)");
261         exit (EXIT_FAILURE);
262     }
263
264     /* STUDENTS CODE */
265     sh->fSt.foodGroup=n;
266     sh->fSt.st.waiterStat = TAKE_TO_TABLE;
267     assignedTable = sh->fSt.assignedTable[n];
268     saveState(nFic, &sh->fst);
269     /* END OF STUDENTS CODE REGION */
270
271     /* exit critical region */
272     if (semUp (semgid, sh->mutex) == -1) {
273         perror ("error on the up operation for mutex semaphore access (WT)");
274         exit (EXIT_FAILURE);
275     }
276
277     if (semUp (semgid, sh->foodArrived[assignedTable]) == -1) {
278         perror ("error on the up operation for foodArrived semaphore access (WT)");
279         exit (EXIT_FAILURE);
280     }
281 }
```

Figura 15 - Função *takeFoodToTable()*

Entidade Repcionista - Ficheiro *semSharedMemReceptionist.c*

Antes de analisarmos as funções desenvolvidas nesta entidade, é necessário apresentar algumas variáveis úteis definidas no início deste ficheiro. É o caso do array **groupRecord[MAXGROUPS]**, que irá guardar o estado de cada **Grupo** na perspetiva do **Repcionista**, útil para decidir que mesa irá ocupar cada um. Foi ainda definida pelo nosso grupo a macro auxiliar **TABLEOCCUPIED**, que irá ser útil para saber se uma mesa está ocupada ou não.

```
45  /* constants for groupRecord */
46  #define TOARRIVE 0
47  #define WAIT      1
48  #define ATTABLE   2
49  #define DONE      3
50
51  /** \brief receptioninst view on each group evolution (useful to decide table binding) */
52  static int groupRecord[MAXGROUPS];
53
54  /* STUDENTS CODE: constant to know if a table is occupied */
55  #define TABLEOCCUPIED -2
```

Figura 16 - Variáveis e Macros auxiliares

- Função **static request waitForGroup()**

Nesta função, o **Repcionista** altera o seu estado para **WAIT_FOR_REQUEST (0)** dentro da **região crítica**, guardando-o. De seguida, espera por um novo pedido por parte de um qualquer **Grupo** através da operação **semDown** do semáforo **receptionistReq**. Depois, entra novamente na **região crítica** para ler o pedido e guardar os dados do mesmo na variável **ret**, que será retornada no final da função. Por fim, sinaliza que está disponível para receber novos pedidos através da operação **semUp** do semáforo **receptionistRequestPossible**.

```
215  static request waitForGroup()
216  {
217      request ret;
218
219      /* enter critical region */
220      if (semDown (semgid, sh->mutex) == -1) {
221          perror ("error on the down operation for mutex semaphore access (RT)");
222          exit (EXIT_FAILURE);
223      }
224
225      /* STUDENTS CODE */
226      sh->fst.st.receptionistStat = WAIT_FOR_REQUEST;
227      saveState(nFic,&sh->fst);
228      /* END OF STUDENTS CODE REGION */
229
230      /* exit critical region */
231      if (semUp (semgid, sh->mutex) == -1) {
232          perror ("error on the up operation for mutex semaphore access (RT)");
233          exit (EXIT_FAILURE);
234      }
235
236      /* STUDENTS CODE */
237      if (semDown (semgid, sh->receptionistReq) == -1) {
238          perror ("error on the down operation for receptionistReq semaphore access (RT)");
239          exit (EXIT_FAILURE);
240      }
241      /* END OF STUDENTS CODE REGION */
```

Figura 17 - Função *waitForGroup()*

```
243     /* enter critical region */
244     if (semDown (semgid, sh->mutex) == -1) {
245         perror ("error on the down operation for mutex semaphore access (RT)");
246         exit (EXIT_FAILURE);
247     }
248
249     /* STUDENTS CODE */
250     ret.reqGroup = sh->fSt.receptionistRequest.reqGroup;
251     ret.reqType = sh->fSt.receptionistRequest.reqType;
252     /* END OF STUDENTS CODE REGION */
253
254     /* exit critical region */
255     if (semUp (semgid, sh->mutex) == -1) {
256         perror ("error on the up operation for mutex semaphore access (RT)");
257         exit (EXIT_FAILURE);
258     }
259
260     /* STUDENTS CODE */
261     if (semUp (semgid, sh->receptionistRequestPossible) == -1) {
262         perror ("error on the up operation for receptionistRequestPossible semaphore access (RT)");
263         exit (EXIT_FAILURE);
264     }
265     /* END OF STUDENTS CODE REGION */
266
267     return ret;
268
269 }
```

Figura 17 - Continuação da função `waitForGroup()`

- Função `static void provideTableOrWaitingRoom(int n)`

Nesta função, o **Repcionista** sabe que lhe foi efetuado um pedido de mesa. Portanto, dentro da **região crítica**, altera o seu estado para **ASSIGNTABLE (1)**, guardando-o. De seguida, ainda dentro da **região crítica**, irá averiguar se existem mesas disponíveis através da chamada da função `decideTableOrWait(n)`, que irá retornar ou o *id* da mesa ou -1, caso o **Grupo** tenha de esperar.

Caso existam mesas livres, o estado do **Grupo** com o *id* **n** na perspetiva do **Repcionista** passa para **ATTABLE (2)**, é atribuída a esse mesmo grupo o *id* da mesa retornado anteriormente, e o estado é guardado. Imediatamente a seguir, é efetuada a operação `setUp` do semáforo `waitForTable[n]`, informando ao **Grupo** com o *id* **n** que pode prosseguir para a mesa atribuída.

Caso não existam mesas livres, o estado do **Grupo** com o id **n** na perspetiva do **Repcionista** passa para **WAIT (1)** e o número de **Grupos** em espera por uma mesa é incrementado em uma unidade, guardando o estado.

Sistemas Operativos - Trabalho Prático 2

```
201 static void provideTableOrWaitingRoom (int n)
202 {
203     /* enter critical region */
204     if (semDown(&semgid, sh->mutex) == -1) {
205         perror ("error on the down operation for mutex semaphore access (RT)");
206         exit (EXIT_FAILURE);
207     }
208
209     /* STUDENTS CODE */
210     sh->fst.st.receptionistStat = ASSIGNTABLE;
211     saveState(nFic, &sh->fst);
212
213     int table_id = decideTableOrWait(n);
214
215     if(table_id != -1)
216     {
217         groupRecord[n] = ATTABLE;
218         sh->fst.assignedTable[n] = table_id;
219         sh->fst.st.receptionistStat = WAIT_FOR_REQUEST;
220         saveState(nFic, &sh->fst);
221
222         if (semUp (&semgid, sh->waitForTable[n]) == -1) {
223             perror ("error on the up operation for waitForTable semaphore access (RT)");
224             exit (EXIT_FAILURE);
225         }
226     }
227     else
228     {
229         groupRecord[n] = WAIT;
230         sh->fst.groupWaiting++;
231         saveState(nFic, &sh->fst);
232     }
233
234     /* END OF STUDENTS CODE REGION */
235
236     /* exit critical region */
237     if (semUp (&semgid, sh->mutex) == -1) {
238         perror ("error on the up operation for mutex semaphore access (RT)");
239         exit (EXIT_FAILURE);
240     }
241 }
```

Figura 17 - Função *provideTableOrWaitingRoom()*

- Função *static int decideTableOrWait(int n)*

Nesta função auxiliar invocada pela função *provideTableOrWaitingRoom* apresentada na página anterior, tem-se como principal objetivo decidir se um grupo deve ocupar uma mesa disponível ou esperar, retornando o *id* da mesa ou **-1**, respetivamente. Sendo uma função de código mais livre, e ainda que no guião disponibilizado na página da disciplina seja dito que apenas existam **2** mesas no **Restaurante**, o nosso grupo decidiu generalizar esta função para um qualquer número de mesas. Para tal, foi inicializado um array de inteiros, **table[NUMTABLES]**, que irá conter os índices de cada mesa do **Restaurante**. De seguida, verificou-se quantos **Grupos** é que estavam a ocupar as mesas através do incremento da variável **tablesOccupied**, assinalando em caso afirmativo a mesa correspondente como ocupada, **TABLEOCCUPIED (-2)**. De seguida, caso a variável **tablesOccupied** fosse diferente do número de mesas do **Restaurante**, saberíamos que pelo menos uma das mesas estaria livre. Portanto, retorna-se a mesa com o **índice mais baixo**. Caso as mesas estejam todas ocupadas, é retornado o valor de **-1**.

```
153 static int decideTableOrWait(int n)
154 {
155     /* STUDENTS CODE */
156     int g, tablesOccupied, i, table[NUMTABLES];
157     for(i=0; i < NUMTABLES; i++)
158         table[i] = i;
159     for(g=0; g < sh->fst.nGroups; g++)
160     {
161         if(groupRecord[g] == ATTABLE)
162         {
163             table[sh->fst.assignedTable[g]] = TABLEOCCUPIED;
164             tablesOccupied++;
165         }
166     }
167
168     if(tablesOccupied != NUMTABLES) {
169         for(i=0; i < NUMTABLES; i++)
170         {
171             if(table[i] != TABLEOCCUPIED)
172             {
173                 return table[i];
174             }
175         }
176     }
177     /* END OF STUDENTS CODE REGION */
178
179     return -1;
180 }
```

Figura 18 - Função *decideTableOrWait()*

- Função **static void receivePayment(int n)**

Nesta função, o **Repcionista** sabe que lhe foi feito um pedido para receber o pagamento de um **Grupo**. Portanto, e dentro da **região crítica**, altera o seu estado para **RECPAY (2)**, guardando-o. De seguida, altera a sua percepção relativamente ao **Grupo** que acabou de efetuar o pedido para **DONE (4)**, guarda o *id* da mesa que acaba de ficar vaga na variável **empty_table** e altera o valor da variável **sh->fSt.assignedTable[n] = -1**, assinalando que o **Grupo** com o *id* **n** já não se encontra na mesa.

De seguida, e ainda dentro da **região crítica**, caso existem **Grupos** à espera que lhes seja atribuída uma mesa, irá ser invocada a função **decideNextGroup**. Esta função irá retornar o *id* mais pequeno do **Grupo** na fila de espera na variável **nextGroup**. De seguida, a percepção do **Repcionista** perante este novo **Grupo** é alterada para **ATTABLE (2)** e é lhe atribuído o *id* da mesa guardado anteriormente na variável **empty_table**. Ao atribuir a mesa, o estado do **Repcionista** é alterado para **WAIT_FOR_REQUEST (0)**, e o número de **Grupos** à espera que lhe seja atribuída uma mesa é decrementado em uma unidade, sendo o estado guardado no final. Imediatamente a seguir, é feita a operação **setUp** do semáforo **waitForTable[nextGroup]**, dando permissão ao **Grupo** em espera para prosseguir para a mesa.

Por fim, e já fora da **região crítica**, o **Grupo** que efetuou o pagamento é informado que este foi completado através da operação **setUp** do semáforo **tableDone[empty_table]** com o *id* da mesa em que se encontrava.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** que requisita o pagamento está sentado, de modo a ser possível realizar a operação **setUp** do semáforo **tableDone[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada fora da região crítica**.

```
331 static void receivePayment (int n)
332 {
333     int nextGroup, empty_table;
334
335     /* enter critical region */
336     if (semDown (semgid, sh->mutex) == -1) {
337         perror ("error on the down operation for mutex semaphore access (RT)");
338         exit (EXIT_FAILURE);
339     }
340
341     /* STUDENTS CODE */
342     sh->fSt.st.receptionistStat = RECPAY;
343     saveState(&fPic, &sh->fSt);
344
345     groupRecord[n] = DONE;
346     empty_table = sh->fSt.assignedTable[n];
347
348     if(sh->fSt.groupsWaiting > 0)
349     {
350         nextGroup = decideNextGroup();
351         groupRecord[nextGroup] = ATTABLE;
352         sh->fSt.assignedTable[nextGroup] = empty_table;
353         sh->fSt.groupsWaiting--;
354         sh->fSt.st.receptionistStat = WAIT_FOR_REQUEST;
355         saveState(&fPic, &sh->fSt);
356         if (semUp (semgid, sh->waitForTable[nextGroup]) == -1) {
357             perror ("error on the up operation for waitForTable semaphore access (RT)");
358             exit (EXIT_FAILURE);
359         }
360     }
361 }
```

Figura 19 - Função *receivePayment()*

```
361     /* END OF STUDENTS CODE REGION */
362
363     /* exit critical region */
364     if (semUp (semgid, sh->mutex) == -1) {
365         perror ("error on the up operation for mutex semaphore access (RT)");
366         exit (EXIT_FAILURE);
367     }
368
369     /* STUDENTS CODE */
370     if (semUp (semgid, sh->tableDone[empty_table]) == -1) {
371         perror ("error on the up operation for tableDone semaphore access (RT)");
372         exit (EXIT_FAILURE);
373     }
374     /* END OF STUDENTS CODE REGION */
375 }
```

Figura 19 - Continuação função *receivePayment()*

- Função *static int decideNextGroup()*

Nesta função auxiliar invocada pela função *receivePayment* apresentada na página anterior, tem-se como principal objetivo decidir qual o próximo **Grupo** a ocupar uma mesa que acabou de ficar livre, caso existam **Grupos** na fila de espera. Para isso, averigua-se qual o estado de cada **Grupo** na percepção do **Repcionista** e, caso se encontre algum no estado **WAIT (1)**, o id desse **Grupo** é retornado pela função.

```
190 static int decideNextGroup()
191 {
192     /* STUDENTS CODE */
193     int g;
194     for(g=0; g < sh->fSt.nGroups; g++)
195     {
196         if(groupRecord[g] == WAIT)
197         {
198             return g;
199         }
200     }
201     /* END OF STUDENTS CODE REGION */
202
203     return -1;
204 }
```

Figura 20 - Função *decideNextGroup()*

Entidade Grupo - Ficheiro *semSharedMemGroup.c*

- Função *goToRestaurant()*

Nesta função, é simulada a ida de cada **Grupo** para o **Restaurante**.

```
152 static void goToRestaurant (int id)
153 {
154     double startTime = sh->fSt.startTime[id] + normalRand(STARTDEV);
155
156     if (startTime > 0.0) {
157         usleep((unsigned int) startTime );
158     }
159 }
```

Figura 21 - Função *goToRestaurant()*

- Função *eat()*

Nesta função, semelhante à anterior, é simulado o tempo que cada **Grupo** demora a comer.

```
168 static void eat (int id)
169 {
170     double eatTime = sh->fSt.eatTime[id] + normalRand(EATDEV);
171
172     if (eatTime > 0.0) {
173         usleep((unsigned int) eatTime );
174     }
175 }
```

Figura 22 - Função *eat()*

- Função `static void checkInAtReception(int d)`

Nesta função, o **Grupo**, ao chegar à receção, averigua se o **Repcionista** se encontra disponível através da operação `semDown` do semáforo `receptionistRequestPossible`. Assim que o **Repcionista** estiver disponível, o **Grupo** altera o seu estado, dentro da **região crítica**, para **ATRECEPTION (2)** e faz um pedido de mesa ao **Repcionista**, **TABLE-REQ (1)**, guardando todas as alterações. Este pedido é efetuado com a operação `semUp` do semáforo `receptionistReq`, já fora da região crítica. Por fim, o **Grupo** fica à espera que lhe seja atribuída uma mesa através da operação `semDown` do semáforo `waitForTable[id]`.

```
187 static void checkInAtReception(int id)
188 {
189     /* STUDENTS CODE */
190     if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
191         perror ("error on the down operation for receptionistRequestPossible semaphore access (GR)");
192         exit (EXIT_FAILURE);
193     }
194     /* END OF STUDENTS CODE REGION */
195
196     /* enter critical region */
197     if (semDown (semgid, sh->mutex) == -1) {
198         perror ("error on the down operation for mutex semaphore access (GR)");
199         exit (EXIT_FAILURE);
200     }
201
202     /* STUDENTS CODE */
203     sh->fst.st.groupStat[id] = ATRECEPTION;
204     sh->fst.receptionistRequest.reqGroup = id;
205     sh->fst.receptionistRequest.reqType = TABLEREQ;
206     saveState(nfic, &sh->fst);
207     /* END OF STUDENTS CODE REGION */
208
209     /* exit critical region */
210     if (semUp (semgid, sh->mutex) == -1) {
211         perror ("error on the up operation for mutex semaphore access (GR)");
212         exit (EXIT_FAILURE);
213     }
214
215     /* STUDENTS CODE */
216     if (semUp (semgid, sh->receptionistReq) == -1) {
217         perror ("error on the up operation for receptionistReq semaphore access (GR)");
218         exit (EXIT_FAILURE);
219     }
220
221     if (semDown (semgid, sh->waitForTable[id]) == -1) {
222         perror ("error on the down operation for mutex semaphore access (GR)");
223         exit (EXIT_FAILURE);
224     }
225     /* END OF STUDENTS CODE REGION */
```

Figura 23 - Função `checkInAtReception()`

- Função **static void orderFood(int id)**

Nesta função, ao chegar à mesa, o **Grupo** averigua se o **Empregado** se encontra disponível através da operação **semDown** do semáforo **waiterRequestPossible**. De seguida, assim que o **Empregado** estiver disponível e dentro da **região crítica**, altera o seu estado para **FOOD_REQUEST (3)** e altera os dados do seu pedido, cujo tipo é de **FOODREQ (3)**, guardando todas as alterações. Após sair da **região crítica**, efetua a operação **semUp** do semáforo **waiterRequest**, assinalando o pedido de comida ao **Empregado**. Por fim, fica à espera que o **Empregado** receba o pedido através da operação **semDown** do semáforo **requestReceived[assignedTable]**.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** que efetua o pedido de comida está sentado, de modo a ser possível realizar a operação **semDown** do semáforo **requestReceived[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada** fora da **região crítica**.

```
239 static void orderFood (int id)
240 {
241     int assignedTable;
242
243     /* STUDENTS CODE */
244     if (semDown (semgid, sh->waiterRequestPossible) == -1) {
245         perror ("error on the down operation for waiterRequestPossible semaphore access (GR)");
246         exit (EXIT_FAILURE);
247     }
248     /* END OF STUDENTS CODE REGION */
249
250     /* enter critical region */
251     if (semDown (semgid, sh->mutex) == -1) {
252         perror ("error on the down operation for mutex semaphore access (GR)");
253         exit (EXIT_FAILURE);
254     }
255
256     /* STUDENTS CODE */
257     sh->fSt.st.groupStat[id] = FOOD_REQUEST;
258     sh->fSt.waiterRequest.reqGroup = id;
259     sh->fSt.waiterRequest.reqType = FOODREQ;
260     assignedTable = sh->fSt.assignedTable[id];
261     saveState(mFic, &sh->fSt);
262     /* END OF STUDENTS CODE REGION */
263
264     /* exit critical region */
265     if (semUp (semgid, sh->mutex) == -1) {
266         perror ("error on the up operation for mutex semaphore access (GR)");
267         exit (EXIT_FAILURE);
268     }
269
270     /* STUDENTS CODE */
271     if (semUp (semgid, sh->waiterRequest) == -1) {
272         perror ("error on the up operation for waiterRequest semaphore access (GR)");
273         exit (EXIT_FAILURE);
274     }
275
276     if (semDown (semgid, sh->requestReceived[assignedTable]) == -1) {
277         perror ("error on the down operation for requestReceived semaphore access (GR)");
278         exit (EXIT_FAILURE);
279     }
```

Figura 24 - Função *orderFood()*

- Função **static void waitFood(int id)**

Nesta função, o **Grupo** já sabe que o **Empregado** recebeu o pedido de comida, e portanto altera o seu estado para **WAIT_FOR_FOOD (4)**, guardando-o dentro da **região crítica**. De seguida, faz a operação **semDown** do semáforo **foodArrived[assignedTable]**, ficando à espera que o **Empregado** lhe traga a comida à mesa. Assim que o **Empregado** entrega a comida à mesa, o **Grupo**, novamente dentro da **região crítica**, altera o seu estado para **EAT (5)**, guardando-o.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** que efetua o pedido de comida está sentado, de modo a ser possível realizar a operação **semDown** do semáforo **foodArrived[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada** **fora da região crítica**.

```
292 static void waitFood (int id)
293 {
294     int assignedTable;
295
296     /* enter critical region */
297     if (semDown (semgid, sh->mutex) == -1) {
298         perror ("error on the down operation for mutex semaphore access (GR)");
299         exit (EXIT_FAILURE);
300     }
301
302     /* STUDENTS CODE */
303     sh->fst.st.groupStat[id] = WAIT_FOR_FOOD;
304     assignedTable = sh->fst.assignedTable[id];
305     saveState(nFic, &sh->fst);
306     /* END OF STUDENTS CODE REGION */
307
308     /* exit critical region */
309     if (semUp (semgid, sh->mutex) == -1) {
310         perror ("error on the up operation for mutex semaphore access (GR)");
311         exit (EXIT_FAILURE);
312     }
313
314     /* STUDENTS CODE */
315     if (semDown (semgid, sh->foodArrived[assignedTable]) == -1) {
316         perror ("error on the down operation for foodArrived semaphore access (GR)");
317         exit (EXIT_FAILURE);
318     }
319     /* END OF STUDENTS CODE REGION */
320
321     /* enter critical region */
322     if (semDown (semgid, sh->mutex) == -1) {
323         perror ("error on the down operation for mutex semaphore access (GR)");
324         exit (EXIT_FAILURE);
325     }
326
327     /* STUDENTS CODE */
328     sh->fst.st.groupStat[id] = EAT;
329     saveState(nFic, &sh->fst);
330     /* END OF STUDENTS CODE REGION */
331
332     /* exit critical region */
333     if (semUp (semgid, sh->mutex) == -1) {
334         perror ("error on the up operation for mutex semaphore access (GR)");
335         exit (EXIT_FAILURE);
336     }
337 }
```

Figura 25 - Função *waitFood()*

- Função **static void checkOutAtReception(int id)**

Nesta função, o **Grupo** averigua se o **Repcionista** se encontra disponível através da operação **semDown** do semáforo **receptionistRequestPossible**. De seguida, assim que o **Repcionista** estiver disponível e dentro da **região crítica**, o **Grupo** altera o seu estado para **CHECKOUT (6)** e altera os dados do pedido ao **Repcionista**, de modo a fazer um pedido de pagamento, **BILLREQ (2)**. Antes de sair da **região crítica**, guarda o estado. De seguida, faz o pedido de pagamento ao **Repcionista** através da operação **semUp** do semáforo **receptionistReq**, e espera que o pagamento seja completado através da operação **semDown** do semáforo **tableDone[assignedTable]**.

Assim que o pagamento for dado como completo, o **Grupo**, dentro da **região crítica**, altera o seu estado para **LEAVING (7)**, guarda-o e sai do **Restaurante**.

Note-se na variável **assignedTable** declarada no início da função. Esta variável irá guardar, dentro da **região crítica**, o índice da mesa em que o **Grupo** que efetua o pedido de comida está sentado, de modo a ser possível realizar a operação **semDown** do semáforo **tableDone[assignedTable]**, fora da **região crítica**. Assim, previne-se a ocorrência de erros durante a execução do programa caso se tentasse aceder à **memória partilhada fora da região crítica**.

```
350 static void checkOutAtReception (int id)
351 {
352     int assignedTable;
353
354     /* STUDENTS CODE */
355     if (semDown (semgid, sh->receptionistRequestPossible) == -1) {
356         perror ("error on the down operation for receptionistRequestPossible semaphore access (GR)");
357         exit (EXIT_FAILURE);
358     }
359     /* END OF STUDENTS CODE REGION */
360
361     /* enter critical region */
362     if (semDown (semgid, sh->mutex) == -1) {
363         perror ("error on the down operation for mutex semaphore access (GR)");
364         exit (EXIT_FAILURE);
365     }
366
367     /* STUDENTS CODE */
368     sh->fSt.st.groupStat[id] = CHECKOUT;
369     sh->fSt.receptionistRequest.reqGroup = id;
370     sh->fSt.receptionistRequest.reqType = BILLREQ;
371     assignedTable = sh->fSt.assignedTable[id];
372     saveState(nFic, &sh->fSt);
373     /* END OF STUDENTS CODE REGION */
374
375     /* exit critical region */
376     if (semUp (semgid, sh->mutex) == -1) {
377         perror ("error on the up operation for mutex semaphore access (GR)");
378         exit (EXIT_FAILURE);
379     }
380 }
```

Figura 26 - Função *checkOutAtReception()*

```
381     /* STUDENTS CODE */
382     if (semUp (semgid, sh->receptionistReq) == -1) {
383         perror ("error on the up operation for receptionistReq semaphore access (GR)");
384         exit (EXIT_FAILURE);
385     }
386
387     if (semDown (semgid, sh->tableDone[assignedTable]) == -1) {
388         perror ("error on the down operation for tableDone semaphore access (GR)");
389         exit (EXIT_FAILURE);
390     }
391     /* END OF STUDENTS CODE REGION */
392
393     /* enter critical region */
394     if (semDown (semgid, sh->mutex) == -1) {
395         perror ("error on the down operation for mutex semaphore access (GR)");
396         exit (EXIT_FAILURE);
397     }
398
399     /* STUDENTS CODE */
400     sh->fSt.st.groupStat[id] = LEAVING;
401     saveState(nFic, &sh->fSt);
402     /* END OF STUDENTS CODE REGION */
403
404     /* exit critical region */
405     if (semUp (semgid, sh->mutex) == -1) {
406         perror ("error on the up operation for mutex semaphore access (GR)");
407         exit (EXIT_FAILURE);
408     }
409
410 }
```

Figura 26 - Continuação da função *checkOutAtReception()*

Assim, dá-se por concluída a explicação de todo o código produzido pelo nosso grupo. De seguida irão ser analisados os **testes efetuados**.

Execução do programa

Para testar todas as funções desenvolvidas, era necessário compilar primeiro cada entidade e testá-la individualmente com os restantes ficheiro binários fornecidos. Para tal, poderíamos usar as regras já existentes no ficheiro *Makefile* presente no diretório **semaphore_restaurant/src**.

```
16  
17 all:      clean group      waiter    chef      receptionist   main clean  
18 gr:       clean group      waiter_bin chef_bin  receptionist_bin main clean  
19 wt:       clean group_bin waiter    chef_bin  receptionist_bin main clean  
20 ch:       clean group_bin waiter_bin chef     receptionist_bin main clean  
21 rt:       clean group_bin waiter_bin chef_bin receptionist   main clean  
22 all_bin:  clean group_bin waiter_bin chef_bin receptionist_bin main clean
```

Figura 27 - Regras presentes no *Makefile*

No final, após termos todas as entidades completas, era necessário fazer o comando “**\$make all**” no terminal. Para executar o programa, é necessário ir até à pasta **semaphore_restaurant/run** através da linha de comandos do terminal e fazer o comando “**./probSemSharedMemRestaurant**”. No entanto, poderíamos também correr o programa **1000 vezes** seguidas utilizando o script em *BASH ./run.sh*. O principal objetivo com estes testes é de verificar se ocorre ou não uma situação de **deadlock**, bem como verificar se os estados das diferentes entidades estão semelhantes aos do output do programa quando se faz o comando “**\$make all_bin**”. Todos os testes efetuados foram gravados em ficheiros com a extensão “.txt” através do comando “**./run.sh > .. tests/test_entidade.txt**”, disponíveis na pasta **semaphore_restaurant/tests**. De seguida, seguem-se duas capturas de ecrã de exemplo ao correr em “**\$make all**”, tanto para 3 como para 16 **Grupos**. O número de **Grupos** pode ser alterado no ficheiro **config.txt**.

Sistemas Operativos - Trabalho Prático 2

```
tiagocm@tiagocm: ~/Documents/Code/CodeUA/projeto-so-02/Tiago/semaphore_restaurant/run$ ./semaphore_restaurant
Run n.º 1000
Restaurant - Description of the internal state

CH WT RC G00 G01 G02 gWT T00 T01 T02
0 0 0 1 1 1 0 . . .
0 0 0 1 1 1 0 . . .
0 0 0 1 1 2 0 . . .
0 0 0 1 1 2 0 . . .
0 0 1 1 1 2 0 . . .
0 0 0 1 1 2 0 . . .
0 0 0 2 1 2 0 . . .
0 0 0 2 1 2 0 . . .
0 0 0 2 1 3 0 . . .
0 0 1 2 1 3 0 . . .
0 0 0 2 1 3 0 1 . .
0 1 0 2 1 3 0 1 . .
0 1 0 2 1 3 0 1 . .
0 1 0 3 1 3 0 1 . .
0 1 0 3 1 4 0 1 . .
1 1 0 3 1 4 0 1 . .
1 0 0 3 1 4 0 1 . .
1 1 0 3 1 4 0 1 . .
1 1 0 4 1 4 0 1 . .
0 1 0 4 1 4 0 1 . .
1 1 0 4 1 4 0 1 . .
1 0 0 4 1 5 0 1 . .
0 0 0 4 1 5 0 1 . .
0 2 0 4 1 5 0 1 . .
0 0 0 4 1 5 0 1 . .
0 0 0 5 1 5 0 1 . .
0 0 0 6 1 5 0 1 . .
0 0 2 6 1 5 0 1 . .
0 0 0 6 1 5 0 . . .
0 0 0 7 1 5 0 . . .
0 0 0 7 2 5 0 . . .
0 0 1 7 2 5 0 . . .
0 0 0 7 2 5 0 . 1 0
0 0 0 7 2 5 0 . 1 0
0 0 0 7 3 5 0 . 1 0
0 1 0 7 3 5 0 . 1 0
0 1 0 7 4 5 0 . 1 0
1 1 0 7 4 5 0 . 1 0
1 0 0 7 4 5 0 . 1 0
0 0 0 7 4 5 0 . 1 0
0 2 0 7 4 5 0 . 1 0
0 2 0 7 5 5 0 . 1 0
0 2 0 7 5 6 0 . 1 0
0 2 2 7 5 6 0 . 1 0
0 2 0 7 5 6 0 . 1 .
0 2 0 7 5 7 0 . 1 .
0 2 0 7 6 7 0 . 1 .
0 2 2 7 6 7 0 . 1 .
0 2 2 7 7 7 0 . . .
```

Figura 28 - Resultado do teste nº 1000 para “\$make all”, com 3 grupos

Sistemas Operativos - Trabalho Prático 2

Figura 29 - Resultado do teste nº 1000 para “\$make all”, com 16 grupos

Figura 29 - Resultado do fim doteste nº 1000 para “\$make all”, com 16 grupos

Conclusão

Após a realização deste segundo trabalho prático, concluímos que os objetivos propostos no guião disponibilizado foram alcançados com sucesso. Todas as funções necessárias para a correta execução da aplicação foram completadas devidamente nas zonas de código previamente assinaladas, não surgindo nenhum **deadlock** durante os testes realizados. No entanto, há a possibilidade de alguns estados das entidades diferirem ligeiramente dos do output quando se executa o programa compilado com “***make all_bin***”.

Com este trabalho, ambos os alunos do grupo fortaleceram os seus conhecimentos em diversos tópicos interessantes como por exemplo a compreensão dos mecanismos associados à execução e sincronização de processos e threads ou a programação concorrente.

É de salientar ainda que o trabalho de equipa e a superação de dificuldades foram fatores importantíssimos no sucesso do trabalho, melhorando as competências interpessoais de ambos os elementos do grupo.

Referências

- Slides disponibilizados na página da disciplina